# Internet Programming Contest

Sponsored by
Duke University
Durham, NC, USA

November 17, 1992

## Welcome

Welcome to the third global Internet programming contest. We at Duke hope that this will be an enjoyable experience. Please do not hesitate to offer suggestions as to how we might improve this contest in the future. Send comments to `khera@cs.duke.edu`.

No participant in the contest should look at these problems prior to 6PM Eastern Standard Time (EST), which is 2300 hours Greenwich Mean Time (GMT), November 17, 1992.

Remember that all input should be read from stdin, that all output to stderr is ignored, and that all output to stdout is judged. If you choose to print prompts (e.g., "Enter a value"), **do NOT print them to stdout**.

There are six (6) problems in this set.

One note to PC users: integers on our machine are 32-bit quantities, so you may wish to use long integers on your machines.

**All submitted programs should have at least one newline at the end of the file or the last line may be truncated by submission scripts.**

The following people contributed significantly in preparation of these problems: Owen Astrachan, David Kotz, Vivek Khera, Lars Nyland, and Steve Tate.

Good luck.

# 1 Problem: Tree Summing

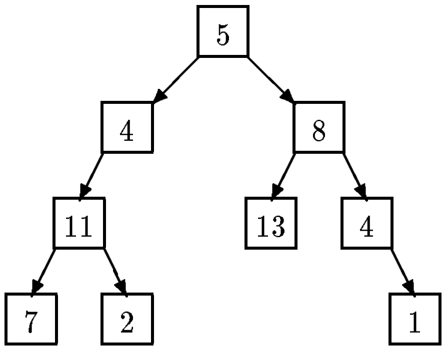
## Background

LISP was one of the earliest high-level programming languages and, with FORTRAN, is one of the oldest languages currently being used. Lists, which are the fundamental data structures in LISP, can easily be adapted to represent other important data structures such as trees.

This problem deals with determining whether binary trees represented as LISP S-expressions possess a certain property.

## The Problem

Given a binary tree of integers, you are to write a program that determines whether there exists a root-to-leaf path whose nodes sum to a specified integer. For example, in the tree shown below there are exactly four root-to-leaf paths. The sums of the paths are 27, 22, 26, and 18.




Binary trees are represented in the input file as LISP S-expressions having the following form.

| | | |
|---|---|---|
| *empty tree* | ::= | () |
| *tree* | ::= | *empty tree* \| (integer *tree tree*) |

The tree diagrammed above is represented by the expression

(5 (4 (11 (7 () ()) (2 () ()) ) ()) (8 (13 () ()) (4 () (1 () ()) ) ) )

Note that with this formulation all leaves of a tree are of the form

(integer () () )

Since an empty tree has no root-to-leaf paths, any query as to whether a path exists whose sum is a specified integer in an empty tree must be answered negatively.

## The Input

The input consists of a sequence of test cases in the form of integer/tree pairs. Each test case consists of an integer followed by one or more spaces followed by a binary tree formatted as an S-expression as described above. All binary tree S-expressions will be valid, but expressions may be spread over several lines and may contain spaces. There will be one or more test cases in an input file, and input is terminated by end-of-file.

### The Output

There should be one line of output for each test case (integer/tree pair) in the input file. For each pair $I, T$ ($I$ represents the integer, $T$ represents the tree) the output is the string *yes* if there is a root-to-leaf path in $T$ whose sum is $I$ and *no* if there is no path in $T$ whose sum is $I$.

### Sample Input

```
22 (5(4(11(7()())(2()()))()) (8(13()())(4()(1()()))))
20 (5(4(11(7()())(2()()))()) (8(13()())(4()(1()()))))
10 (3
     (2 (4 () () )
        (8 () () ) )
     (1 (6 () () )
        (4 () () ) ) )
5 ()
```

### Sample Output

```
yes
no
yes
no
```

## 2 Problem: Power of Cryptography

### Background

Current work in cryptography involves (among other things) large prime numbers and computing powers of numbers modulo functions of these primes. Work in this area has resulted in the practical use of results from number theory and other branches of mathematics once considered to be of only theoretical interest.

This problem involves the efficient computation of integer roots of numbers.

### The Problem

Given an integer $n \geq 1$ and an integer $p \geq 1$ you are to write a program that determines $\sqrt[n]{p}$, the positive $n^{\text{th}}$ root of $p$. In this problem, given such integers $n$ and $p$, $p$ will always be of the form $k^n$ for an integer $k$ (this integer is what your program must find).

### The Input

The input consists of a sequence of integer pairs $n$ and $p$ with each integer on a line by itself. For all such pairs $1 \leq n \leq 200$, $1 \leq p < 10^{101}$ and there exists an integer $k$, $1 \leq k \leq 10^9$ such that $k^n = p$.

### The Output

For each integer pair $n$ and $p$ the value $\sqrt[n]{p}$ should be printed, i.e., the number $k$ such that $k^n = p$.

### Sample Input

```
2
16
3
27
7
4357186184021382204544
```

### Sample Output

```
4
3
1234
```

# 3 Problem : Simulation Wizardry

## Background

Simulation is an important application area in computer science involving the development of computer models to provide insight into real-world events. There are many kinds of simulation including (and certainly not limited to) discrete event simulation and clock-driven simulation. Simulation often involves approximating observed behavior in order to develop a practical approach.

This problem involves the simulation of a simplistic *pinball* machine. In a pinball machine, a steel ball rolls around a surface, hitting various objects (*bumpers*) and accruing points until the ball "disappears" from the surface.

## The Problem

You are to write a program that simulates an idealized pinball machine. This machine has a flat surface that has some obstacles (bumpers and walls). The surface is modeled as an $m \times n$ grid with the origin in the lower-left corner. Each bumper occupies a grid point. The grid positions on the edge of the surface are walls. Balls are shot (appear) one at a time on the grid, with an initial position, direction, and lifetime. In this simulation, all positions are integral, and the ball's direction is one of: up, down, left, or right. The ball bounces around the grid, hitting bumpers (which accumulates points) and walls (which does not add any points). The number of points accumulated by hitting a given bumper is the *value* of that bumper. The speed of all balls is one grid space per timestep. A ball "hits" an obstacle during a timestep when it would otherwise move on top of the bumper or wall grid point. A hit causes the ball to "rebound" by turning right (clockwise) 90 degrees, without ever moving on top of the obstacle and without changing position (only the direction changes as a result of a rebound). Note that by this definition sliding along a wall does not constitute "hitting" that wall.

A ball's lifetime indicates how many time units the ball will live before disappearing from the surface. The ball uses one unit of lifetime for each grid step it moves. It also uses some units of lifetime for each bumper or wall that it hits. The lifetime used by a hit is the *cost* of that bumper or wall. As long as the ball has a positive lifetime when it hits a bumper, it obtains the full score for that bumper. Note that a ball with lifetime one will "die" during its next move and thus cannot obtain points for hitting a bumper during this last move. Once the lifetime is non-positive (less than or equal to zero), the ball disappears and the game continues with the next ball.

## The Input

Your program should simulate one game of pinball. There are several input lines that describe the game. The first line gives integers $m$ and $n$, separated by a space. This describes a cartesian grid where $1 \leq x \leq m$ and $1 \leq y \leq n$ on which the game is "played". It will be the case that $2 < m < 51$ and $2 < n < 51$. The next line gives the integer cost for hitting a wall. The next line gives the number of bumpers, an integer $p \geq 0$. The next $p$ lines give the $x$ position, $y$ position, value, and cost, of each bumper, as four integers per line separated by space(s). The $x$ and $y$ positions of all bumpers will be in the range of the grid. The value and cost may be any integer (i.e., they may be negative; a negative cost *adds* lifetime to a ball that hits the bumper). The remaining lines of the file represent the balls. Each line represents one ball, and contains four integers separated by space(s): the initial $x$ and $y$ position of the ball, the direction of movement, and its lifetime. The position will be in range (and not on top of any bumper or wall). The direction will be one of four values: 0 for increasing $x$ (right), 1 for increasing $y$ (up), 2 for decreasing $x$ (left), and 3 for decreasing $y$ (down). The lifetime will be some positive integer.

### The Output

There should be one line of output for each ball giving an integer number of points accumulated by that ball in the same order as the balls appear in the input. After all of these lines, the total points for all balls should be printed.

### Sample Input

```
4 4
0
2
2 2 1 0
3 3 1 0
2 3 1 1
2 3 1 2
2 3 1 3
2 3 1 4
2 3 1 5
```

### Sample Output

```
0
0
1
2
2
5
```

# 4 Problem: Climbing Trees

## Background

Expression trees, B and B* trees, red-black trees, quad trees, PQ trees; trees play a significant role in many domains of computer science. Sometimes the name of a problem may indicate that trees are used when they are not, as in the Artificial Intelligence planning problem traditionally called the *Monkey and Bananas problem.* Sometimes trees may be used in a problem whose name gives no indication that trees are involved, as in the *Huffman code.*

This problem involves determining how pairs of people who may be part of a "family tree" are related.

## The Problem

Given a sequence of *child-parent* pairs, where a pair consists of the child's name followed by the (single) parent's name, and a list of query pairs also expressed as two names, you are to write a program to determine whether the query pairs are related. If the names comprising a query pair are related the program should determine what the relationship is. Consider academic advisees and advisors as exemplars of such a single parent genealogy (we assume a single advisor, i.e., no co-advisors).

In this problem the child-parent pair $p$ $q$ denotes that $p$ is the child of $q$. In determining relationships between names we use the following definitions:

- $p$ is a *0-descendent* of $q$ (respectively *0-ancestor*) if and only if the child-parent pair $p$ $q$ (respectively $q$ $p$) appears in the input sequence of child-parent pairs.
- $p$ is a *k-descendent* of $q$ (respectively *k-ancestor*) if and only if the child-parent pair $p$ $r$ (respectively $q$ $r$) appears in the input sequence and $r$ is a $(k-1)$-descendent of $q$ (respectively $p$ is a $(k-1)$-ancestor of $r$).

For the purposes of this problem the relationship between a person $p$ and a person $q$ is expressed as exactly one of the following four relations:

1. child — grand child, great grand child, great great grand child, *etc.*

   By definition $p$ is the "child" of $q$ if and only if the pair $p$ $q$ appears in the input sequence of child-parent pairs (i.e., p is a 0-descendent of q); $p$ is the "grand child" of $q$ if and only if $p$ is a 1-descendent of $q$; and

   $$p \text{ is the ``}\underbrace{\text{great great} \ldots \text{great}}_{n \text{ times}}\text{grand child'' of } q$$

   if and only if $p$ is an $(n+1)$-descendent of $q$.

2. parent — grand parent, great grand parent, great great grand parent, *etc.*

   By definition $p$ is the "parent" of $q$ if and only if the pair $q$ $p$ appears in the input sequence of child-parent pairs (i.e., $p$ is a 0-ancestor of $q$); $p$ is the "grand parent" of $q$ if and only if $p$ is a 1-ancestor of $q$; and

   $$p \text{ is the ``}\underbrace{\text{great great} \ldots \text{great}}_{n \text{ times}}\text{grand parent'' of } q$$

   if and only if $p$ is an $(n+1)$-ancestor of $q$.

3. cousin — $0^{\text{th}}$ cousin, $1^{\text{st}}$ cousin, $2^{\text{nd}}$ cousin, *etc.*; cousins may be once removed, twice removed, three times removed, *etc.*

   By definition $p$ and $q$ are "cousins" if and only if they are related (i.e., there is a path from $p$ to $q$ in the implicit undirected parent-child tree). Let $r$ represent the least common ancestor of $p$ and $q$ (i.e., no descendent of $r$ is an ancestor of both $p$ and $q$), where $p$ is an $m$-descendent of $r$ and $q$ is an $n$-descendent of $r$.

   Then, by definition, cousins $p$ and $q$ are "$k^{\text{th}}$ cousins" if and only if $k = \min(n, m)$, and, also by definition, $p$ and $q$ are "cousins removed $j$ times" if and only if $j = \mid n - m \mid$.

4. sibling — $0^{\text{th}}$ cousins removed 0 times are "siblings" (they have the same parent).

## The Input

The input consists of parent-child pairs of names, one pair per line. Each name in a pair consists of lower-case alphabetic characters or periods (used to separate first and last names, for example). Child names are separated from parent names by one or more spaces. Parent-child pairs are terminated by a pair whose first component is the string "*no.child*". Such a pair is NOT to be considered as a parent-child pair, but only as a delimiter to separate the parent-child pairs from the query pairs. There will be no circular relationships, i.e., no name $p$ can be *both* an ancestor and a descendent of the same name $q$.

The parent-child pairs are followed by a sequence of query pairs in the same format as the parent-child pairs, i.e., each name in a query pair is a sequence of lower-case alphabetic characters and periods, and names are separated by one or more spaces. Query pairs are terminated by end-of-file.

There will be a maximum of 300 different names overall (parent-child and query pairs). All names will be fewer than 31 characters in length. There will be no more than 100 query pairs.

## The Output

For each query-pair $p$ $q$ of names the output should indicate the relationship $p$ *is-the-relative-of* $q$ by the appropriate string of the form

- child, grand child, great grand child, great great ...great grand child
- parent, grand parent, great grand parent, great great ...great grand parent
- sibling
- $n$ cousin removed $m$
- no relation

If an $m$-cousin is removed 0 times then only $m$ *cousin* should be printed, i.e., *removed 0* should NOT be printed. Do not print *st, nd, rd, th* after the numbers.

### Sample Input

```
alonzo.church oswald.veblen
stephen.kleene alonzo.church
dana.scott alonzo.church
martin.davis alonzo.church
pat.fischer hartley.rogers
mike.paterson david.park
dennis.ritchie pat.fischer
hartley.rogers alonzo.church
les.valiant mike.paterson
bob.constable stephen.kleene
david.park hartley.rogers
no.child no.parent
stephen.kleene bob.constable
hartley.rogers stephen.kleene
les.valiant alonzo.church
les.valiant dennis.ritchie
dennis.ritchie les.valiant
pat.fischer michael.rabin
```

### Sample Output

```
parent
sibling
great great grand child
1 cousin removed 1
1 cousin removed 1
no relation
```

# 5 Problem: Unidirectional TSP

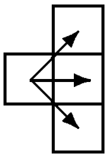
## Background

Problems that require minimum paths through some domain appear in many different areas of computer science. For example, one of the constraints in VLSI routing problems is minimizing wire length. The Traveling Salesperson Problem (TSP) — finding whether all the cities in a salesperson's route can be visited exactly once with a specified limit on travel time — is one of the canonical examples of an NP-complete problem; solutions appear to require an inordinate amount of time to generate, but are simple to check.

This problem deals with finding a minimal path through a grid of points while traveling only from left to right.

## The Problem

Given an $m \times n$ matrix of integers, you are to write a program that computes a path of minimal weight. A path starts anywhere in column 1 (the first column) and consists of a sequence of steps terminating in column $n$ (the last column). A step consists of traveling from column $i$ to column $i+1$ in an adjacent (horizontal or diagonal) row. The first and last rows (rows 1 and $m$) of a matrix are considered adjacent, i.e., the matrix "wraps" so that it represents a horizontal cylinder. Legal steps are illustrated below.



The *weight* of a path is the sum of the integers in each of the $n$ cells of the matrix that are visited.

For example, two slightly different $5 \times 6$ matrices are shown below (the only difference is the numbers in the bottom row).

| | | | | | |
|---|---|---|---|---|---|
| 3 | 4 | 1 | 2 | 8 | 6 |
| 6 | 1 | 8 | 2 | 7 | 4 |
| 5 | 9 | 3 | 9 | 9 | 5 |
| 8 | 4 | 1 | 3 | 2 | 6 |
| 3 | 7 | 2 | 8 | 6 | 4 |

| | | | | | |
|---|---|---|---|---|---|
| 3 | 4 | 1 | 2 | 8 | 6 |
| 6 | 1 | 8 | 2 | 7 | 4 |
| 5 | 9 | 3 | 9 | 9 | 5 |
| 8 | 4 | 1 | 3 | 2 | 6 |
| 3 | 7 | 2 | 1 | 2 | 3 |

The minimal path is illustrated for each matrix. Note that the path for the matrix on the right takes advantage of the adjacency property of the first and last rows.

## The Input

The input consists of a sequence of matrix specifications. Each matrix specification consists of the row and column dimensions in that order on a line followed by $m \cdot n$ integers where $m$ is the row dimension and $n$ is the column dimension. The integers appear in the input in row major order, i.e., the first $n$ integers constitute the first row of the matrix, the second $n$ integers constitute the second row and so on. The integers on a line will be separated from other integers by one or more spaces. Note: integers are not restricted to being positive. There will be one or more matrix specifications in an input file. Input is terminated by end-of-file.

For each specification the number of rows will be between 1 and 9 inclusive; the number of columns will be between 1 and 100 inclusive. No path's weight will exceed integer values representable using 30 bits.

### The Output

Two lines should be output for each matrix specification in the input file, the first line represents a minimal-weight path, and the second line is the cost of a minimal path. The path consists of a sequence of $n$ integers (separated by one or more spaces) representing the rows that constitute the minimal path. If there is more than one path of minimal weight the path that is lexicographically smallest should be output.

### Sample Input

```
5 6
3 4 1 2 8 6
6 1 8 2 7 4
5 9 3 9 9 5
8 4 1 3 2 6
3 7 2 8 6 4
5 6
3 4 1 2 8 6
6 1 8 2 7 4
5 9 3 9 9 5
8 4 1 3 2 6
3 7 2 1 2 3
2 2
9 10 9 10
```

### Sample Output

```
1 2 3 4 4 5
16
1 2 1 5 4 5
11
1 1
19
```

# 6 Problem: The Postal Worker Rings Once

## Background

Graph algorithms form a very important part of computer science and have a lineage that goes back at least to Euler and the famous *Seven Bridges of Königsberg* problem. Many optimization problems involve determining efficient methods for reasoning about graphs.

This problem involves determining a route for a postal worker so that all mail is delivered while the postal worker walks a minimal distance, so as to rest weary legs.

## The Problem

Given a sequence of streets (connecting given intersections) you are to write a program that determines the minimal cost tour that traverses every street at least once. The tour must begin and end at the same intersection.

The "real-life" analogy concerns a postal worker who parks a truck at an intersection and then walks all streets on the postal delivery route (delivering mail) and returns to the truck to continue with the next route.

The cost of traversing a street is a function of the length of the street (there is a cost associated with delivering mail to houses and with walking even if no delivery occurs).

In this problem the number of streets that meet at a given intersection is called the *degree* of the intersection. There will be at most two intersections with odd degree. All other intersections will have even degree, i.e., an even number of streets meeting at that intersection.

## The Input

The input consists of a sequence of one or more postal routes. A route is composed of a sequence of street names (strings), one per line, and is terminated by the string "*deadend*" which is NOT part of the route. The first and last letters of each street name specify the two intersections for that street, the length of the street name indicates the cost of traversing the street. All street names will consist of lowercase alphabetic characters. For example, the name *foo* indicates a street with intersections $f$ and $o$ of length 3, and the name *computer* indicates a street with intersections $c$ and $r$ of length 8. No street name will have the same first and last letter and there will be at most one street directly connecting any two intersections. As specified, the number of intersections with odd degree in a postal route will be at most two. In each postal route there will be a path between all intersections, i.e., the intersections are connected.

## The Output

For each postal route the output should consist of the cost of the minimal tour that visits all streets at least once. The minimal tour costs should be output in the order corresponding to the input postal routes.

### Sample Input

```
one
two
three
deadend
mit
dartmouth
linkoping
tasmania
york
emory
cornell
duke
kaunas
hildesheim
concord
arkansas
williams
glasgow
deadend
```

### Sample Output

```
11
114
```